Известный физик Ричард Фейнман как то произнес, ставшую крылатой фразу: «Если вы думаете, что понимаете квантовую механику, значит, вы её не понимаете». Что-то похожее можно сказать и о первом знакомстве с парадигмой объектно-ориентированного программирования (ООП): «Если в самом начале вы решили, что осознали ООП, то на самом деле вы его еще не осознали». Почему же часто так сложно сделать первые шаги в этом направлении и начать писать программы в новом, современном стиле? В целом, ответ прост. ООП – это именно новая **парадигма** программирования, то есть, новый, иной способ мышления при написании программ. А научить себя мыслить иначе, не так, как прежде, - в этом и состоит главная сложность.

А что было прежде? Большинство языков программирования (ЯП) позволяют составлять программы в виде последовательности команд (набора определенных операторов). Такой подход получил название **императивное программирование**. (От слова «императив» - приказ, команда.) И, например, изучая язык Си, мы придерживались именно этой императивной парадигмы. То есть, способ мышления состоял в выборе последовательности команд, которые в процессе выполнения приводили к требуемому результату. Например, в этой парадигме легко составить программу вычисления суммы значений элементов какого-либо массива:

```cpp
int ar[] = {1, 2, 3, 4};
int s = 0;
for(int i = 0; i < sizeof(ar) / sizeof(*ar); ++i)
    s += ar[i];
```

Как правило, сложности в понимании самой парадигмы, как набора команд, у начинающего программиста не возникает. Наверное, потому что мы, буквально, вырастаем в среде, где постоянно раздаются различные команды. И позже, перейти к другой парадигме, другой схеме мышления, оказывается не так просто.

Тем не менее, общая концепция ООП достаточно проста. Нужно лишь к ней привыкнуть и использовать во всей полноте. И мы сейчас с вами сделаем первый шаг для «перепрошивки» наших мозгов, чтобы уйти от чистого императивного программирования в сторону ООП.

Объекты и классы

Начнем с того, что часто, решаемую задачу, можно разложить на относительно независимые блоки (подзадачи), которые, взаимодействуя между собой, достигают исходной поставленной цели. Давайте образно представим часы, как задачу проекта. Условно ее можно разложить на объекты: «циферблат», «маятниковый часовой механизм», «корпус». Эти объекты, согласованно взаимодействуя между собой, образуют единое устройство – маятниковые часы. И я здесь произнес первое важное слово – **объект**. Что такое объект в концепции ООП? В большинстве ЯП каждый объект располагается в независимой области памяти и содержит определенные данные. Например, объект «циферблат» может иметь:

```cpp
unsigned radius; // размер циферблата
unsigned color; // цвет циферблата
```

И так далее. Все эти свойства объектов программист придумывает сам. В результате, может формироваться множество объектов, отличающихся между собой этими количественными характеристиками, образуя единый **класс** таких объектов. В парадигме ООП именно класс отвечает за порождение объектов определенного вида. Например, можно описать класс круглых циферблатов:

```cpp
class RoundDial {
    type_digits type; // тип чисел на циферблате (арабские, римские)
    unsigned color; // цвет циферблата
};
```

А, затем, создавать объекты этого класса:

```cpp
RoundDial rd1, rd2, rd3;
```

То есть, класс в ООП выступает в роли схемы, чертежа, по которому конструируются объекты этого класса. Программисту достаточно объявить нужный класс, чтобы, затем, формировать множество объектов соответствующего типа. И так для всех объектов, используемых в программе.

Методы класса

Также с каждым объектом, помимо данных, могут быть еще связаны специальные функции, именуемые **методами**. Как правило, именно через них происходит взаимодействие с объектами. Например, в классы можно было бы добавить следующие методы:

```cpp
class RoundDial {
    unsigned radius; // размер циферблата
    unsigned color; // цвет циферблата

public:
    void update();
    void set_color(unsigned);
    void set_radius(unsigned);
};
…
```

В результате каждый объект класса имеет свой независимый набор данных и общий набор методов, через которые происходит взаимодействие с этим объектом. То есть, каждый объект превращается в активный элемент программы, в котором выполняется логика работы алгоритма, описанная в классе этого объекта. Таким образом, **класс выступает независимой единицей программного кода, в котором реализован определенный фрагмент**. Например, класс RoundDial берет на себя всю логику обработки и отображения круглых часовых циферблатов. Класс PClockWork – логику работы часового механизма. А класс GroundClock обеспечивает функционирование часов, как единое целое. То есть, **классы – это не просто хранилища данных, а полноценные фрагменты программы, работающие, по возможности, независимо друг от друга и остальной части программы**. Причем работа алгоритма класса, как правило, выполняется на уровне объектов, так как именно объекты содержат конкретные данные, которые можно обрабатывать тем или иным способом (алгоритмом). А в классе лишь объявляются данные для объектов и порядок их обработки на уровне методов. То есть, к**ласс нужно воспринимать, как схему, чертеж, по которой формируются и работают его объекты.**

Инкапсуляция

Но раз класс образует единую и независимую программную единицу со своей внутренней логикой работы, то должен существовать механизм, который бы «защищал» класс от внешних некорректных действий, нарушающих целостность его работы? И такой механизм в ООП, конечно же, есть. Он получил название **инкапсуляции** и часто реализуется через ограничение доступа к переменным и методам класса. Например, в классе PClockWork все переменные размещаются в закрытой (приватной) секции. И доступ к ним возможен только изнутри объекта или внутри методов при их вызове. Напрямую, извне доступ к этим данным ограничен. А вот методы, через которые происходит взаимодействие, должны объявляться как открытые

(публичные), чтобы иметь к ним доступ извне. То есть, **определяя тот или иной класс, программист разрешает взаимодействие с ним через публичные разрешенные методы и реже напрямую через данные**. Соответственно, методы должны быть построены так, чтобы их вызов не нарушал целостность данных объекта и логику его работы, обеспечивая, тем самым, защиту объектов класса.

Наследование и полиморфизм

А теперь самое главное, что ООП делает по-настоящему новой парадигмой программирования, а не просто описанием программы на уровне независимых объектов. Новые классы можно определять на основе других, ранее сформированных классов. В концепции ООП это называется **наследованием** классов. При этом классы, от которых происходит наследование, получили название **базовых**, а новые сформированные – **дочерних**.

На первый взгляд, ничего особенного. Просто, еще один способ объявления новых классов. Но не спешите с выводами. Базовые и дочерние классы образуют такой симбиоз, который выводит ООП на совершенно новый уровень проектирования программ. Я покажу это на примере наших часов.

Смотрите, всю логику взаимодействия между объектами можно описать на общем уровне – уровне базовых классов, так как в них объявлены все необходимые для этого переменные и методы. А конкретная реализация будет определяться дочерними классами. Причем, методы в базовых классах можно объявлять так, чтобы реализации для них подставлялись из соответствующих дочерних классов. В концепции ООП это называется **полиморфизмом**. В результате, **для одного набора дочерних классов мы получаем одно программное решение, а для другого набора – совершенно другое**. При этом в будущем, нам будет достаточно определить какие-либо новые дочерние классы, чтобы они автоматически встраивались в общую логику работы программы и давали качественно новый результат. Именно качественно новый, так как классы – это, фактически, реализации отдельных алгоритмов. Поэтому, меняя дочерние классы, мы меняем на алгоритмическом уровне поведение соответствующих частей программы. И это потрясающий способ организации программного кода, который дает нам парадигма ООП, основанная на трех базовых элементах:

- инкапсуляции – обеспечение целостности данных и их обработки;
- наследовании – образование одних классов из других;

- полиморфизма – взаимодействие с тем или иным дочерним классом через базовый класс.

Заключение

Ничего подобного классический императивный подход к программированию нам дать не может. Конечно, если программа небольшая и не сложная, то писать ее с использованием ООП нет никакого смысла. Но реальные проекты средней и более высокой сложности без применения ООП сейчас практически невозможны. Объектно-ориентированный подход позволяет нам относительно просто создавать по-настоящему модульные программы, описывать логику на общем уровне, и достаточно просто расширять функционал, добавляя новые дочерние классы.